# Evolutionary Algorithms: Final report

Sander Prenen (r0701014)

December 5, 2020

## 1 Formal requirements

The report is structured for fair and efficient grading of over 100 individual projects in the space of only a few days. Please respect the exact structure of this document. You are allowed to remove sections 1 and 7. Brevity is the soul of wit: a good report will be **around** 7 **pages** long. The hard limit is 10 pages.

It is recommended that you use this LaTeX template, but you are allowed to reproduce it with the same structure in a WYSIWYG-editor. The purple text containing our evaluation criteria can be removed. You should replace the blue text with your discussion. **The questions we ask in blue are there to guide which topics to discuss**, rather than an exact list of questions that must be answered. Feel free to add more items to discuss.

This report should be uploaded to Toledo by January 4, 2021 at 16:00 CET. It must be in the **Portable Document Format** (pdf) and must be named `r0123456_intermediate.pdf`, where r0123456 should be replaced with your student number.

## 2 Metadata

- **Group members during group phase:** Karel Everaert and Matthias Maeyens
- **Time spent on group phase:** 15 hours
- **Time spent on final code:** 40 hours
- **Time spent on final report:** 10 hours

## 3 Modifications since the group phase

**Goal:** Based on this section, we will evaluate insofar as you are able to analyse common problems arising in the design and implementation of evolutionary algorithms and your ability to effectively solve them.

### 3.1 Main improvements

List the main changes that you implemented since the group phase. You do not need to explain the employed techniques in detail; for this, you should refer to the appropriate subsection of section 4 of the report.

**Initialization of the population:** A greedy heuristic was used to initialize the population in order to start from a fitter population than the random population used in the group phase.

**Schemes:** Different schemes were carried out for all the main operators in the evolutionary algorithm (selection, mutation, recombination, elimination).

**Implementation of a local search operator:** A local search operator is implemented to improve the candidate solutions and speed up the convergence.

**Parallelization of the most expensive parts of the algorithm:** Computationally expensive parts of the algorithm like the local search operator have been parallelized by using the `multiprocessing` library.

### 3.2 Issues resolved

Recall the list of issues from the group phase. Describe how you solved these issues in the individual phase.

**Mutation:** The mutation operator used in the group phase had little on larger problem sizes. A number of different mutation operators were implemented to cope with this problem.

**Elimination scheme:** The algorithm created in the group phase had the problem that the fittest candidate solution got mutated unwillingly, making divergence from the optimal tour possible. This issue was fixed by integrating an elitism elimination scheme to always keep the fittest individual in the next generation.

**Diversity promotion:** The implementation in the group phase suffered from a lack of diversity. This caused the algorithm to converge to local optima. This problem was solved by adding a diversity promotion scheme to the evolutionary algorithm.

**Recombination operator:** The recombination operator used in the group phase did not preserve the order of the nodes in the tour, but rather tried to preserve the position of the nodes. Several new recombination operators were created to deal with this problem.

## 4 Final design of the evolutionary algorithm

**Goal:** Based on this section, we will evaluate insofar as you are able to design and implement an advanced, effective evolutionary algorithm for solving a model problem.

In this section, you should describe all components of your final evolutionary algorithm and how they fit together.

### 4.1 Representation

How do you represent the candidate solutions? What is your motivation to choose this one? What other options did you consider? How did you implement this specifically in Python (e.g., a list, set, numpy array, etc)?

An individual is represented as a 1D `numpy` array with the order of the visited nodes. A `numpy` array is used rather than a plain python list, because of the more efficient manner in which these arrays get stored in memory and the vectorized operations that are available in the `numpy` library. A population is represented as a two dimensional array (each row is an individual). This choice was made for the same reasons as mentions above.

### 4.2 Initialization

How do you initialize the population? How did you determine the number of individuals? Did you implement advanced initialization mechanisms (local search operators, heuristic solutions)? If so, describe them. Do you believe your approach maintains sufficient diversity? How do you ensure that your population enrichment scheme does not immediately take over the population? Did you implement other initialization schemes that did not make it to the final version? Why did you discard them? How did you determine the population size?

The population is initialized by using a greedy algorithm. The algorithm starts in a node and picks the nearest not visited node as the next node. The starting points are decided at random. This is a variation on all nearest neighbors (ANN) as proposed by Kaabi and Harrath [1]. This heuristic method creates less diversity than random initialization. Therefore a diversity promotion scheme is implemented and described in section 4.8. The random initialization method is still available by setting the boolean value of `use_random_initialization` to true. This version explores more of the search space, but it will never reach the heuristic solution for the bigger problems in the given time span.

### 4.3 Selection operators

Which selection operators did you implement? If they are not from the slides, describe them. Can you motivate why you chose this one? Are there parameters that need to be chosen? Did you use an advanced scheme to vary these parameters throughout the iterations? Did you try other selection operators not included in the final version? Why did you discard them?

The following selection operators were implemented: k-tournament selection, linear rank based selection and roulette wheel selection with geometric decaying selection pressure. Some of these selection operators depend on parameters chosen by the user. For k-tournament this parameter is $k$, the number of individuals to be randomly selected for the tournament. The roulette wheel selection depends on the starting selection pressure and the decay for this selection pressure. The linear rank based selection does not need any parameters to be chosen.
The final version of the evolutionary algorithm uses the roulette wheel selection with geometric decay. By doing a hyperparameter search, as described in section 4.11, the values of the starting selection pressure and the decay are set to x and y respectively. The generation of individuals in this operator is not done using inverse transform sampling, as this would require $\mathcal{O}(n)$ time per sample. Instead the Vose's alias method is used as explained by Schwarz [2, 3]. This method is able to sample in constant time, but uses $\mathcal{O}(n)$ time to initialize. This initialization can be reused to select all the individuals during the current generation.
A fitness proportionate selection scheme was briefly introduced but discarded due to the problems mentioned in the slides.

### 4.4 Mutation operators

Which mutation operators did you implement? If they are not from the slides, describe them. How do you choose among several mutation operators? Do you believe it will introduce sufficient randomness? Can that be controlled with parameters? Do you use self-adaptivity? Do you use any other advanced parameter control mechanisms (e.g., variable across iterations)? Did you try other mutation operators not included in the final version? Why did you discard them?

The mutation operator that existed in the group phase, the sequence swap operator, was simplified to a simple swap operator that swaps two nodes at random. This operator had little effect on larger problem instances due to its proportionally small range. Therefore three different operators were introduced: reverse sequence mutation (RSM), partial shuffle mutation (PSM) and a hybrid of the two called hybridizing PSM RSM mutation (HPRM) [4].
RSM chooses two nodes in the individual at random and reverses the sequence between the two nodes. PSM is similar but randomly shuffles the sequence between the nodes, thus introducing more randomness. HPRM is a combination of the two where the sequence between the nodes is reversed element-wise and shuffled after each reversion.
All the operators discussed cannot be controlled by a parameter to tune the amount of randomness they introduce. The final evolutioanry algorithm uses RSM because it proved the best in the numerical experiments. The operator is applied with a probability $\alpha$ to an individual, with $\alpha = 0.2 * \frac{\textit{mean objective}}{\textit{mean objective} + \textit{best objective}}$. This idea is based on the idea from Kaabi and Harrath [1].

### 4.5 Recombination operators

Which recombination operators did you implement? If they are not from the slides, describe them. How do you choose among several recombination operators? Why did you choose these ones specifically? Explain how you believe that these operators can produce offspring that combine the best features from their parents. How does your operator behave if there is little overlap between the parents? Can your recombination be controlled with parameters; what behavior do they change? Do you use self-adaptivity? Do you use any other advanced parameter control mechanisms (e.g., variable across iterations)? Did you try other recombination operators not included in the final version? Why did you discard them? Did you consider recombination with arity strictly greater than 2?

The recombination operator implemented during the group phase, partially mapped crossover (PMX), was kept. Two other recombination operators called order crossover (OX) and sequential constructive crossover (SCX) proposed by Davis [5] and Ahmed [6]. All three operators start from the same principle, two crossover points are chosen at random in both parents. The nodes between these points are then copied to the offspring. How the rest of the nodes are copied differs between the operators. PMX tries to keep the positions of the nodes consistent by using cycles in the parents. OX copies the unused nodes from the parent of which the nodes between the crossover points have not been copied to the child. SCX searches in both parents to the next unused node and picks the one that is the nearest to the node previously inserted in the child. This operator thus only creates one child from two parents. Therefore double the number of recombinations are needed in order to generate the same number of offspring.
Since OX and SCX respect the relative order of nodes in the parents, these will probably perform better than PMX on the asymmetric TSP. The experiments performed in section 4.11 also show this. The final algorithm uses OX since it had better results on bigger problem instances. This operator guarantees that if an edge between two nodes is present in both parents, it will definitely be copied to the offspring. If there is little overlap between the parents, then the operator will practically copy half of both parents to the children. Due to the complexity of recombination operators on permutations operators with arity greater than two have not been considered. The operators themselves don't depend on any parameters. They just create the number of children specified in the `offsping_size` variable.

### 4.6 Elimination operators

Which elimination operators did you implement? If they are not from the slides, describe them. Why did you select this one? Are there parameters that need to be chosen? Did you use an advanced scheme to vary these parameters throughout the iterations? Did you try other elimination operators not included in the final version? Why did you discard them?

The only elimination scheme implemented is $(\lambda+\mu)$-elimination, which was created in the group phase. Since it is not guaranteed that the offspring will be better than the previous generation, this is the best option. To make sure that the best objective value will not diverge, elitism is applied to the population after elimination. This is described in section 4.12.

### 4.7 Local search operators

What local search operators did you implement? Describe them. Did they cause a significant improvement in the performance of your algorithm? Why (not)? Did you consider other local search operators that did not make the cut? Why did you discard them? Are there parameters that need to be determined in your operator? Do you use an advanced scheme to determine them (e.g., adaptive or self-adaptive)?

Three different local search operators where considered. All three operators are based on a neighborhood structure. The first one is a 3-opt method where a tour is seen as three different arcs (A-B-C). If the alternative A-C-B is shorter then this tour is kept. The second operator is a 2-opt operator. This operator swaps two neighboring nodes in an individual if it is shorter than the other way around. The last operator reverses the individual as a whole if it is shorter. This operator was not implemented since it would not have a big effect because of the way the distance matrix is generated.
The final algorithm uses the 3-opt local search operator based on the one proposed by Kanellakis and Papadimitriou [7] and improved by Bentley [8] and Nagata and Soler [9]. Instead of splitting the tour in arcs in all possible places, a list of nearest neighbors is used for each node. Once a first splitting point is chosen, then the second point is limited by this list of nearest neighbors. This reduces the computations needed to perform the local search while still receiving a good improvement on the individual. To reduce the computations even further, the first splitting point is chosen at random. This makes that there is not always a better individual found using the operator, but if the individual makes it to the next generation then another random splitting point will be chosen. This approach proved experimentally better than doing a full 3-opt search on the individuals.
The only parameter that needs to be chosen is the number of nearest neighbors that is kept in the neighbor list. Nagata and Soler suggested to use 15. This is also the value used in this evolutionary algorithm.

### 4.8 Diversity promotion mechanisms

Did you implement a diversity promotion scheme? If yes, which one? If no, why not? Describe the mechanism you implemented. In what sense does the mechanism improve the performance of your evolutionary algorithm? Are there parameters that need to be determined? Did you use an advanced scheme to determine them?

Two different diversity promotion schemes have been implemented. Fitness sharing in the elimination phase and the removal of duplicate individuals after elimination as proposed by Herrera-Poyatos and Herrera [10].
Fitness sharing is implemented as described in the slides of this course. The distance function used is the number of edges that are different between two individuals. The computation of the distance function is very expensive and is therefore executed in parallel. This scheme needs a maximum distance $\sigma$ that will be considered in order to apply the penalties.
The removal of duplicate individuals is based on the objective values. If the objective values are not equal then the individuals cannot be the same. This check is much easier than the distance computation in fitness sharing. If two or more individuals are the same, then one is kept and the others are replaced by a greedy randomized algorithm. The individual is initialized starting from a random node and the next chosen node is picked randomly from the set restricted candidates. These candidates are all nodes that are less than $\sigma$ percent longer than the greedy option.
The final algorithm uses the greedy diversification option because it performed better on the larger problem instances. Computing the distances for all individuals proved to be too costly. The parameter $\sigma$ used in the greedy diversification algorithm is set to 10 as suggested by the authors of the paper.

### 4.9 Stopping criterion

Which stopping criterion did you implement? Did you combine several criteria?

During the development phase, a combination of two criteria was used. The first one was a limit on the number of generations (500 in this case) and the second one was the number of generations in which the best solutions had not changed. The former criterion was obsolete after for the larger problems due to the computationally expensive local search operator. Since the elitism scheme was used in the final version of the algorithm, the stopping criterion has been dropped in order to fully use the given time span.

### 4.10 The main loop

Describe the main loop of your evolutionary algorithm using a clear picture (preferred) or high-level pseudocode. In what order do you apply the various operators? Why that order? If you are using several selection, mutation, recombination, elimination, and local search operators, describe how you choose among the possibilities. Are you selecting/eliminating all individuals in parallel, or one by one? With or without replacement?

Before the main loop starts, some parameters that depend on the tour size are initialized. The offspring is generated in the `recombination` function. In there the individuals are selected with replacement in parallel.

Table 1: Parameter values for the hyperparameter search

| Parameter | Values |
|---|---|
| $\lambda$ | 8, 16, 32, 100 |
| $\mu$ | 8, 16, 32, 50, 100 |
| $l$ | 0, 0.25, 0.5, 0.75, 1 |

Afterwards all parents and children are supplied to the `mutation` function. This function applies mutation with a given probability $\alpha$ specified in section 4.4. Then the local search operator is applied on half of these individuals. Elimination and diversification is then applied to the optimized population. The next part is 'restarting' the algorithm if there is a risk of settling in a local minimum. Finally the results of this generation is reported to the reporter and the loop is restarted. The main loop can be seen in Figure 1.

### 4.11 Parameter selection

For all of the parameters that are not automatically determined by adaptivity or self-adaptivity (as you have described above), describe how you determined them. Did you perform a hyperparameter search? How did you do this? How did you determine these parameters would be valid both for small and large problem instances?

Tuning all parameters of an evolutionary algorithm is huge problem on its own. Therefore not all possible combinations of the parameters and operators were tested. The tuning process was split in multiple parts in order to decrease the time needed. The first part of this process was deciding the mutation and recombination operators out of the possibilities described in section 4. All the different combinations of mutation and recombination operators were tested 20 times on the smallest tour. The most promising mutation operator were RSM and HPRM, but there was not a recombination operator that was significantly better. The remaining mutation operators and all recombination operators were then tested on `tour194.csv` in order to try create a more significant difference between the recombination operators. The experiment showed that OX and SCX performed similar whilst PMX performed a fraction worse. For the final evolutionary algorithm OX was chosen, because it typically needed less generations than SCX to converge. Note that during this experiment it was assumed that all operators had similar performance with the same parameters. This is normally not the case.

Now that the operators are chosen, different experiments can be performed on the parameters of the evolutionary algorithm. If a value of a parameter was suggested by the authors of a paper about the operator, then this value was used and not changed or included in the hyperparameter search. The following parameters were selected to perform a hyperparameter search on: the population size $\lambda$, the offspring size $\mu$ and the percentage of individuals to perform local search on $l$. In Table 1 all the tested values for the different parameters can be found. The hyperparameter search was performed on the smallest problem instance `tour29`. Each combination was tested 8 times to reduce the random effect of the operators. Since the mean values were not much of an indicator for the performance, the success rate was used. Where success is defined as finding the global minimum tour length. Obviously bigger population and offspring sizes show a higher succes rate, as seen in Figure 2,but there ain't no such thing as a free lunch [11]. Therefore promising values that demand less calculations, like a population size around 25 and an offspring size around 50, are explored in a new hyperparameter search on a larger problem instance.

### 4.12 Other considerations

Did you consider other items not listed above, such as elitism, multiobjective optimization strategies (e.g., island model, pareto front approximation), a parallel implementation, or other interesting computational optimizations (e.g. using advanced algorithms or data structures)? You can describe them here or add additional subsections as needed.

Some operators, like the local search operator and the fitness sharing elimination, are computationally expensive. But the results of these operators don't depend on each other. Therefore these operators can be executed in parallel. To accomplish this a `RawArray` from `multiprocessing` is used for the distance matrix. This is a one dimensional array that cannot be locked, but this is not needed because the parallel processes will not alter the matrix.

Sometimes during the execution of the algorithm, it occurs that it gets stuck in a local minimum and the diversity and local search operator are not able to get the individual out of it. To cope with this problem, the evolutionary algorithm 'restarts' itself by randomly initializing a completely new population when the best individual has not been changed for a set number of iterations.

```
    def optimize(self, filename: str):
        """
        The main loop of the genetic algorithm.
        :param filename: The filename of the tour for which a candidate solution needs to be
                                                                found.
        """
        # Read the distance matrix from file
        data = np.loadtxt(filename, delimiter=",")
        self.tour_size = data.shape[0]
        self.raw_distance_matrix = RawArray(ctypes.c_double, self.tour_size * self.tour_size)
        self.distance_matrix = np.frombuffer(self.raw_distance_matrix, dtype=np.float64).reshape(
                                                                self.tour_size,
                                                                                                self
```

```
        np.copyto(self.distance_matrix, data)

        self.sigma = np.floor(0.05 * self.tour_size)
        self.build_nearest_neighbor_list()
        self.init_dictionary()

        population = self.initialize_population()

        while not self.is_converged():
            offspring = self.recombination(population)
            mutated_population = self.mutation(population, offspring)

            # mutated_population = self.local_search(mutated_population)
            optimized_population = self.local_search_parallel(mutated_population)

            # population, scores = self.fitness_sharing_elimination(optimized_population)

            population, scores = self.elimination(optimized_population)
            population, scores = self.eliminate_duplicate_individuals(population, scores)
            population, scores = self.elitism(population, scores)

            self.update_scores(population[0], scores)

            if self.same_best_objective % 20 == 0 and self.same_best_objective != 0:
                if self.same_best_objective > 40:
                    break
                self.use_random_initialization = True
                population = self.initialize_population()
                population[0] = self.best_solution

            if self.mean_objective != np.inf:
                self.alpha = 0.2 * self.mean_objective / (self.best_objective + self.
                                                                mean_objective)

            # Call the reporter with:
            #  - the mean objective function value of the population
            #  - the best objective function value of the population
            #  - a 1D numpy array in the cycle notation containing the best solution
            #    with city numbering starting from 0
            time_left = self.reporter.report(self.mean_objective, self.best_objective, self.
                                                            best_solution)
            self.time_left = time_left
            if time_left < 0:
                break

        return 0
```

Figure 1: The main loop of the evolutionary algorithm

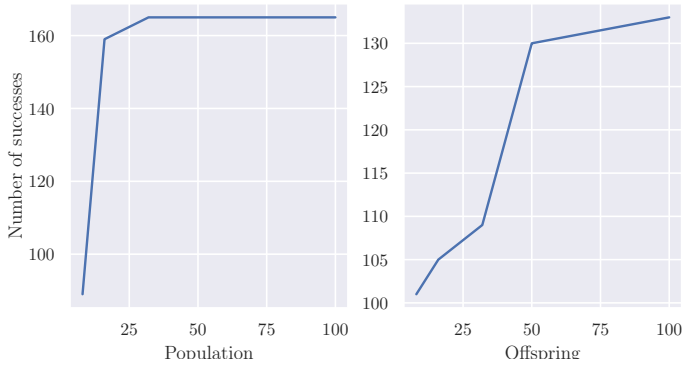


Figure 2: Successes for different parameter values on `tour29`.

Table 2: Static parameters used in the numerical experiments

| Parameter | Value | Parameter | Value |
|---|---|---|---|
| Population size $\lambda$ | | Offspring size $\mu$ | |
| Number of nearest neighbors | 15 | Selection pressure | 0.99 |
| Margin $\sigma$ in greedy diversification | 0.10 | Selection pressure decay | 0.99 |

Elitism is used to stop the algorithm from diverging from the optimal value. If the best individual after elimination is worse than the best individual of the previous generation, then this individual is inserted and the current worst individual is discarded.

## 5 Numerical experiments

**Goal:** Based on this section and our execution of your code, we will evaluate the performance (time, quality of solutions) of your implementation and your ability to interpret and explain the results on benchmark problems.

### 5.1 Metadata

What parameters are there to choose in your evolutionary algorithm? Which fixed parameter values did you use for all experiments below? If some parameters are determined based on information from the problem instance (e.g., number of cities), also report their specific values for the problems below.

Report the main characteristics of the computer system on which you ran your evolutionary algorithm. Include the processor or CPU (including the number of cores and clock speed), the amount of main memory, and the version of Python 3.

All experiments conducted in this section are done using the same parameter values. The parameters that have a fixed value can be found in Table 2 and the dynamic parameters take the values specified in section 4

All numeric experiments were performed on a 64-bit version of Ubuntu 20.04 with a quad-core Intel i5-7300U @ 2.60 GHz and 16 GB of memory. Python version 3.8.5 was used.

### 5.2 tour29.csv

Run your algorithm on this benchmark problem (with the 5 minute time limit from the Reporter). Include a typical convergence graph, by plotting the mean and best objective values in function of the time (for example based on the output of the Reporter class).

What is the best tour length you found? What is the corresponding sequence of cities?

Interpret your results. How do you rate the performance of your algorithm (time, memory, speed of convergence, diversity of population, quality of the best solution, etc)? Is your solution close to the optimal one?

Solve this problem 1000 times and record the results. Make a histogram of the final mean fitnessess and the final best fitnesses of the 1000 runs. Comment on this figure: is there a lot of variability in the results, what are the means and the standard deviations?

### 5.3 tour100.csv

Run your algorithm on this benchmark problem (with the 5 minute time limit from the Reporter). Include a typical convergence graph, by plotting the mean and best objective values in function of the time (for example based on the output of the Reporter class).

What is the best tour length you found in each case?

Interpret your results. How do you rate the performance of your algorithm (time, memory, speed of convergence, diversity of population, quality of the best solution, etc)? Is your solution close to the optimal one?

### 5.4 tour194.csv

Run your algorithm on this benchmark problem (with the 5 minute time limit from the Reporter). Include a typical convergence graph, by plotting the mean and best objective values in function of the time (for example based on the output of the Reporter class).

What is the best tour length you found?

Interpret your results. How do you rate the performance of your algorithm (time, memory, speed of convergence, diversity of population, quality of the best solution, etc)? Is your solution close to the optimal one?

### 5.5 tour929.csv

Run your algorithm on this benchmark problem (with the 5 minute time limit from the Reporter). Include a typical convergence graph, by plotting the mean and best objective values in function of the time (for example based on the output of the Reporter class).

What is the best tour length you found?

Interpret your results. How do you rate the performance of your algorithm (time, memory, speed of convergence, diversity of population, quality of the best solution, etc)? Is your solution close to the optimal one?

Did your algorithm converge before the time limit? How many iterations did you perform?

## 6 Critical reflection

**Goal:** Based on this section, we will evaluate your understanding and insight into the main strengths and weaknesses of your evolutionary algorithms.

Describe the main lessons learned from this project. What do you think are the main strong points of evolutionary algorithms in general? Did you apply these strengths in this project? What are the main weaknesses of evolutionary algorithms and of your implementation in particular? Do you think these can be avoided or mitigated? How? Do you believe evolutionary algorithms are appropriate for this problem? Why (not)? What surprised you and why? What did you learn from this project?

## 7 Other comments

In case you think there is something important to discuss that is not covered by the previous sections, you can do it here.

## References


[1] Jihene Kaabi and Youssef Harrath. Permutation rules and genetic algorithm to solve the traveling salesman problem. *Arab Journal of Basic and Applied Sciences*, 26(1):283–291, 2019.

[2] Keith Schwarz. Darts, dice, and coins: Sampling from a discrete distribution. `https://www.keithschwarz.com/darts-dice-coins`, 2011. Accessed: 2020-11-13.

[3] Michael D. Vose. A linear algorithm for generating random numbers with a given distribution. *Software Engineering, IEEE Transactions on*, 1991.

[4] Abdoun Otman, Chakir Tajani, and Jaafar Abouchabka. A new mutation operator for solving an np-complete problem: Travelling salesman problem. 05 2012.

[5] Lawrence Davis. Applying adaptive algorithms to epistatic domains. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'85, page 162–164, San Francisco, CA, USA, 1985. Morgan Kaufmann Publishers Inc.

[6] Zakir Ahmed. Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator. *International Journal of Biometric and Bioinformatics*, 3, 03 2010.

[7] Paris-C. Kanellakis and Christos H. Papadimitriou. Local search for the asymmetric traveling salesman problem. *Operations Research*, 28(5):1086–1099, 1980.

[8] Jon Louis Bentley. Experiments on traveling salesman heuristics. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, page 91–99, USA, 1990. Society for Industrial and Applied Mathematics.

[9] Yuichi Nagata and David Soler. A new genetic algorithm for the asymmetric traveling salesman problem, expert systems with applications. *Expert Systems with Applications*, 39(10):8947–8953, 2012.

[10] Andrés Herrera-Poyatos and Francisco Herrera. Genetic and memetic algorithm with diversity equilibrium based on greedy diversification. *CoRR*, abs/1702.03594, 2017.

[11] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.